基于事务映射区间求交的高效频繁模式挖掘算法 *

吴　磊[a]，程良伦[a]，王　涛[b]

(广东工业大学 a. 计算机学院; b. 自动化学院，广州 510006)

摘　要：关联规则挖掘是数据挖掘重要研究课题，大数据处理对关联规则挖掘算法效率提出了更高要求，而关联规则挖掘的最耗时的步骤是频繁模式挖掘。针对当前频繁模式挖掘算法效率不高的问题，结合 Apriori 算法和 FP-growth 算法，提出一种基于事务映射区间求交的频繁模式挖掘算法 IITM（interval interaction and transaction mapping），只需扫描数据集两次来生成 FP 树，然后扫描 FP 树将每个项的 ID 映射到区间中，通过区间求交来进行模式增长。该算法解决了 Apriori 算法需要多次扫描数据集，FP-growth 算法需要迭代地生成条件 FP 树来进行模式增长而带来的效率下降的问题。在真实数据集上的实验显示，在不同的支持度下 IITM 算法都要要优于 Apriori、FP-growth 以及 PIETM 算法。

关键词 ：数据挖掘；频繁模式；事务映射；区间求交

中图分类号：TP301

Efficient frequent pattern mining algorithm based on interval interaction and transaction mapping

Wu Lei[a], Cheng Lianglun[a], Wang Tao[b]

(*a. Faculty of Computer*, *b. Faculty of Automation Guangdong University of Technology*, *Guangzhou* 510006, *China*)

Abstract: Association rules mining is an important research topic in data mining. Big data processing puts forward higher requirements for the efficiency of association rules mining algorithm, where the most time consuming step is frequent pattern mining. For the problem that the state of art frequent pattern mining algorithm is not efficient, a frequent pattern mining algorithm based on interval interaction and transaction mapping (IITM) is proposed, which combines Apriori algorithm and FP-growth algorithm. This algorithm just needs to scan the dataset twice to generate the FP tree, and then scan the FP tree to map the ID of each transaction to the interval. It growths the frequent pattern by interval interaction and solves the problem that the Apriori algorithm needs to scan the dataset multiple times, the FP-growth algorithm needs to iterate to generate the conditional FP tree, which reduce the efficiency of the frequent pattern mining. Experiments on real dataset show that the IITM algorithm is superior to Apriori, FP-growth, and PIETM algorithms at different support.

Key Words: data mining; frequent pattern; interval interaction; transaction mapping

0　引言

关联规则挖掘是数据挖掘的一个重要的部分，它旨在从大规模数据当中挖掘出隐藏的关系并提供决策支持。关联规则挖掘最早是 Agrawal 在文献[1]中提出的，它使用关联规则挖掘来在商场购物篮数据当中提取出有用的信息，决策者根据这些信息来制定商场的销售策略。如今关联规则挖掘已经被应用到了各个领域如网站数据[2]、硬件系统监测[3]、车辆和通信数据 [4]等。

随着应用的逐渐扩展，关联规则挖掘面对的数据量越来越大，这就对关联规则挖掘算法的效率提出的更高的要求。影响关联规则挖掘效率的一个最重要的步骤是频繁模式挖掘，它的目的是发现数据当中多次出现在同一条记录当中的数据项。目前国内外已经有了很多关于频繁模式挖掘算法的研究，其中以 Apriori 算法[1]和 FP-Growth[5]算法最具代表性，其他的算法大多数是基于这两个算法改进的。其中 Apriori 算法通过频繁 n-1 项集来生成候选 n 项集，对于每个候选 n 项集，它都需要扫描整个数据集来获取它的支持度，从而判断该候选项集是否需要被剪枝，当数据量非常大的时候多次扫描数据集就导致该算法

基金项目：智能制造物联网的数据感知；传输和海量数据处理；国家基金广东省联合基金重点项目；（U1201251）；面向船舶产品的智能制造集成平台研究及产业化；广东省省级科技计划项目；（2016B090918045）制造物联网协同感知的服务组合优化模型与寻优算法研究；国家自然科学基金青年科学基金项目（61502110）

作者简介：吴磊（1993-），男，硕士研究生，主要研究方向为数据挖掘，机器学习（1104505174@qq.com）；程良伦（1965-），男，博士，博导，主要研究方向为网络控制与系统集成，网络与信息化控制，物联网与物理信息融合系统；王涛（1983-），博士，主要研究方向为传感网与物联网，制造物联，物理信息融合系．

的效率低下。FP-Growth 算法则只需要扫描两次数据集来生成频繁模式树，然后通过迭代地生成条件模式树来生成频繁项集，当频繁模式树非常大的时候迭代生成条件模式树的开销会非常大。文献[6]提出了 PIETM 算法，它首先生成 FP 树，然后将 FP 树当中的各项投影到区间当中，使用类似 Apriori 算法的方法，用频繁 n-1 项集合并生成候选 n 项集，并且使用生成该 n 项集的两个 n-1 项集所对应的区间进行求并来生成 n 项集的区间集，接着使用容斥原理来获取候选项集的支持度。但是使用容斥原理需要获取候选项的所有的真子集，然后在支持度表当中获取它们的支持度。当候选项较长，频繁项的数量较多的时候该算法的消耗还是非常大。

本文在 PIETM 算法的基础之上提出 IITM（interval interaction and transaction mapping）算法，IITM 算法参考 PIETM 算法的事务映射思想将 FP 树当中的项映射到区间当中，与 PIETM 算法不同的是 IITM 算法采用频繁 n-1 项集和频繁 1 项集结合的方式来生成候选 n 项集，并且使用求交而不是求并的方式来生成候选 n 项集的区间集，IITM 算法能够直接从生成的候选项集的区间当中获取其支持度而不用通过容斥原理来获取。本文还根据 IITM 算法的特点设计了一种快速定位项集对应的区间集的存储结构，以及一种高效的区间求交的方法。本文在四个真实数据集上对 IITM, PIETM,FP-growth,Apriori 算法进行比较，IITM 算法在不同的支持度下都要优于其他算法。

1 相关工作

国内外已经提出的很多频繁模式挖掘算法。Apriori 算法首先扫描一次数据集来生成超过最小支持度的频繁 1-项集。然后通过频繁 1 项集合并生成候选 2-项集，为了验证候选 2-项集的支持度是否超过最小支持度需要再次扫描数据集。Apriori 的优点是实现简单，但是需要多次扫描数据集，因此有很多方法被提出来，来优化该算法。比如文献[7]当中提出的 DHP 算法来通过一个哈希表结构来加速候选项集的生成。但没能解决影响 Apriori 算法性能的根本问题，需要进行多次的数据集扫描。在文献[8]当中提出了一种 Inter-Apriori 算法，该算法对数据集当中的所有的事务进行编号，只需扫描一次数据集来得到各项频繁项集出现的事务的编号，此后无须再扫描数据集。由频繁 n-1-项集生成频繁 n 项集时只需合并对应的频繁 n-1-项集的事务编号集合即可。但当数据集非常大的时候采用这种算法需要存储的编号过多，对内存的消耗特别大。在文献[9,10]在 MapReduce 框架当中实现了 Apriori 的并行化算法，在处理海量数据时有一定优势，但仍然没有触动 Apriori 算法的本质缺陷。文献[11]中提出了 UT-Miner 算法，该算法主要针对稀疏数据，使用一种单元三元组的数组结构来存储数据集当中的项和事务的关系以提高挖掘效率。同时 Apriori 也扩展到了其他 的场景如不确定数据集中的加权频繁模式挖掘[12]等。

FP-Growth 算法只需扫描两次数据集，第一次扫描获取频繁 1-项集，第二次将每条事务中的非频繁项去掉，将剩下的项按支持度递减排序然后插入 FP 树当中。然后采用自底向上方式生成条件模式树来生成频繁项集，但是当数据集较大的时候，生成频繁模式树的消耗会非常大。因此有很多算法被提出来改进该算法，在文献[13]当中提出了 IFP-Growth 算法，IFP-Growth 提出一种新的树结构 $FP\text{-}tree^{+}$在 FP-tree 上加上地址信息来降低生成的条件模式树的数量，从而加快了挖掘的率，但是该算法和 FP-Growth 算法相比需要存储更多的信息，因此需要消耗更多的内存。在文献[14]中提出了 $FP\text{-}Growth^{*}$算法，在该算法当中使用 FP 数组来存储模式信息来有效地消除非频繁模式从而提高挖掘效率，同样该算法在提高了效率的同时会加大内存的消耗。同时 FP-Grwoth 也被用于在其他的应用场景如最大频繁模式挖掘[15],增量频繁模式挖掘[16],最近加权频繁模式挖掘[17],不确定数据集上的频繁模式挖掘文献[18],Spark 环境下的频繁模式挖掘[19]。上述基于 Apriori 的改进算法大多继承了 Apriori 算法需要多次扫描数据集的缺点，基于 FP-Growth 的改进算法大多需要在原始的 FP-tree 的基础上加上额外的存储信息需要消耗大量的内存。本文针对以上问题提出 IITM 算法，该算法只需扫描两次数据集来生成 FP-tree,然后扫描 FP-tree 将事务的 id 映射到高度压缩的区间当中。接下来使用这些区间来生成候选项集的区间来获取支持度。

2 基本概念

2.1 问题定义

首先，对问题进行定义，频繁模式挖掘的定义如下：设 $I=\{i_1,i_2,i_3,\ldots,i_n\}$代表项的集合，T 为 I 中的项组成的集合，称 T 为一条事务。一个数据集 D 由一系列的事务组成，I 的一个子集 $X\subseteq T$ 称为一个项集，如果 X 当中包含 k 个项，那么称 X 为 k-项集。一个项集的支持度表示为 sup(X),它代表包含 X 的事务的数量占总事务数的百分比。用户指定一个最低的支持度 minSup，频繁模式挖掘的目的是找出所有的 $sup(X)\geqslant minSup$ 的 X。

2.2 FP 树

FP 树是一种用来存储事务信息的紧凑的树型数据结构。构造 FP 树时首先需要扫描一次数据集来获取所有的 1 项集的支持度。接着第二次扫描数据集，将每条事务当中的非频繁项去掉，将剩下的项按支持度递减排序后插入 FP 树当中。初始时 FP 树结构当中只有一个根节点 root，将排序好的各事务依次插入到根节点下。设排序好的事务为 T[p,P],其中 p 为事务当中的第一个项，P 为余下的项,nodeP 为存储项 p 的节点。插入函数为 insert(T,Node),其中 T 为要插入的事务，node 为事务 T 要插入的节点。如果节点 node 有子节点 nodeP 则将 nodeP 的计数加 1，如果 Node 没有子节点 nodeP 则为 node 新建一个子节点 nodeP,将其计数信息设为 1.如果 P 不为空则递归调用 insert(P,p)。如果 P 为空则 FP 树生成结束。使用表 1 当中的示例数据生成 FP 树如图 1。

表 1 示例数据

原始事务	各项的支持度	去掉非频繁项按计数递减排序
A,D,B	D:5 E:4 A:4 B:3 C:2 G:1	D,A,B
B,C,E		E,B
A,B,D		D,A,B
A,C,D,E		D,A,E
A,D,E		D,A,E
D,E,G		D,E

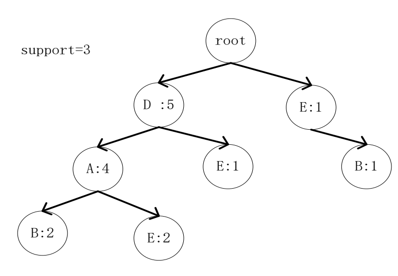


图 1 FP 树的构造过程

3 IITM 算法

本文提出新的 IITM 算法，该算法参考 PIETM 算法[6]中的事务映射技术，改进候选项集的生成、候选项集的支持度的获取的方式以及区间的存储结构。

3.1 事务映射

IITM 算法[6]使用事务映射技术，并对其结构进行改进以适应 IITM 算法的模式增长过程。事务映射将 FP 树当中的各个节点所在事务编号映射到区间当中，这样在进行模式增长时就不需要迭代生成条件模式树，而是直接使用对应的区间就能够获取候选项集的支持度信息。在 FP 树构造完成之后，遍历一次 FP 树来生成区间信息，区间是一个存储整形数据的线性表结构，区间的结构为[*s*,*e*]其中 *s* 代表区间的开始编号，*e* 代表区间的结束编号。每个节点的区间和节点的计数 *x* 相联系,满足关系 $x=s+e-1$。IITM 算法采用深度优先遍历 FP 树的方式来为每个节点生成区间，区间的生成过程满足以下条件：

a)若节点为其父节点的第一个子节点，如果其父节点为根节点，那么该节点 *s*=1，如果不为根节点，那么该节点 *s*=father.s;

b)若节点不是其父节点的第一个子节点，那么该节点 *s*=prevBrother.e+1；

c)对所有的节点 e=s+count-1；

d)father.s≤s≤e≤father.e;

其中:count 代表节点的支持度计数，father 代表当前节点的父节点，prevBroteher 代表当前节点的上一个兄弟节点。

名称相同的节点的区间储存到同一个区间列表当中，区间列表的结构为{support;s_1,e_1;s_2,e_2;...;s_n,e_n}其中 support 代表该区间列表集合对应的项集的支持度，其值满足 $support=e_1-s_1+1+e_2-s_2+1+...+e_n-s_n+1$。根据图 1 当中的 FP 树为每个节点生成的区间及区间列表如图 2、表 2 所示。

IITM 算法需要多次查询已生成的频繁项集以及其对应的区间列表信息，因此将频繁项集和其对应的区间采用键值对<key,value>的形式来进行储存。其中 key 储存频繁项集，value 储存其对应的区间。为了提高查询的效率，引入 hash 表结构（本文称该表为项集哈希索引表），根据频繁项集的 hash 值将这些键值对的结构存入项集哈希索引表索引的项集区间储存表当中，当需要查询某个频繁项集对应的区间列表的时候，首先根据频繁项集计算 hash 值，然后根据 hash 值去项集哈希索引表当中查询区间表的位置，最后去项集哈希索引表上找到相应的区间列表。又由于 IITM 算法使用频繁 n-1 项集和频繁 1 项集结合来生成候选 n 集，因此算法当中需要多次查询候选 n-1 项集，为了加快查询的速度将不同长度的候选项集及其对应的区间放入不同的 hash 表当中，然后在一个线性表当中储存这些 hash 表的引用。储存结构如图 3 所示，当数据集当中的频繁模式较多的时候，采用这种储存结构能够极大地加快搜索已经挖掘出的频繁模式的以及其对应的区间列表效率。

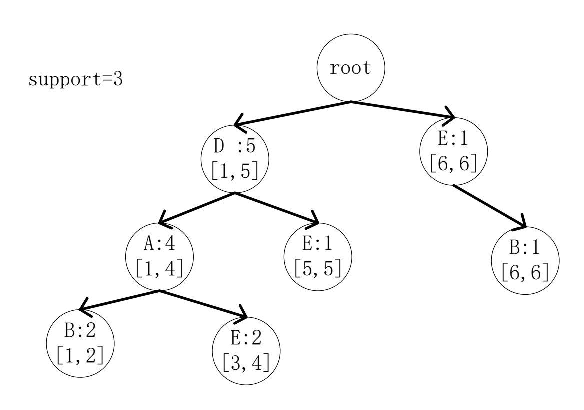


图 2 FP 树当中构造的区间的示意图

表 2 根据图 1 中的 FP 树构造的区间列表

项名	区间计数	区间
D	5	[1,5]
E	4	[3,4][5,5],[6,6]
A	4	[1,4]
B	3	[1,2],[6,6]

当生成频繁 1 项集的区间列表的时候，首先初始化频繁 1 项集的项集哈希索引表，然后在项集长度索引表当中的编号为 1 的位置生成对频繁 1 项集的项集哈希索引表的引用，接着扫描 FP 树为每个节点生成区间，每生成一个区间，计算该节点对应项的 hash 值，根据该 hash 值查找到对应的区间储存表，查看表当中是否存在该项的区间记录，如果不存在，则为该项在区间索引表当中生成区间列表，并将该项的区间插入到区间表当中。如果存在则将该项的区间插入到已有的区间列表当中，因为采用的是深度优先遍历，所以后生成的同名项的区间 *s*,*e* 的值总是要大于先生成的区间的 *s*,*e* 值，因此插入时按下列的方式:

设已有的区间列表为{support;s_1,e_1;s_2,e_2;…; s_{i-1},e_{i-1}},待插入的区间为[s_i,e_i],根据区间的生成规则可知 $s_i \geq e_{i-1}$,如果 $s_i=e_{i-1}$ 则插入后的区间列表为{support+e_i-s_i+1;s_1,e_1; s_2,e_2;…;s_{i-1},e_i},若 $s_i>e_{i-1}$ 则插入后的区间列表为{support+e_i-s_i+1 ;s_1,e_1;s_2, e_2;… ; s_{i-1}, e_{i-1};s_i,e_i}。

根据 FP 树生成区间的算法的伪代码如下：

```
输入：FP 树根节点 root，区间初始值 s，起始时传入的 s 值为 1
输出：项集长度索引表 ILIT
generateIntervals(root,s)
if root == null
  return
if ILIT ==null
  new ILIT//项集长度索引表不存在，新建一个
childs = root.childs;//获取 root 节点的子节点
for node in childs//遍历所有孩子节点
  interval=(s,s+node.count-1)//生成节点的区间
  save interval into ILIT//将区间保存到 ILIT 索引的指定的位置的当中
  generateIntervals(node, s);//递归为子节点生成区间
  s = s + node.count;//更新区间初始值
output ILIT
```

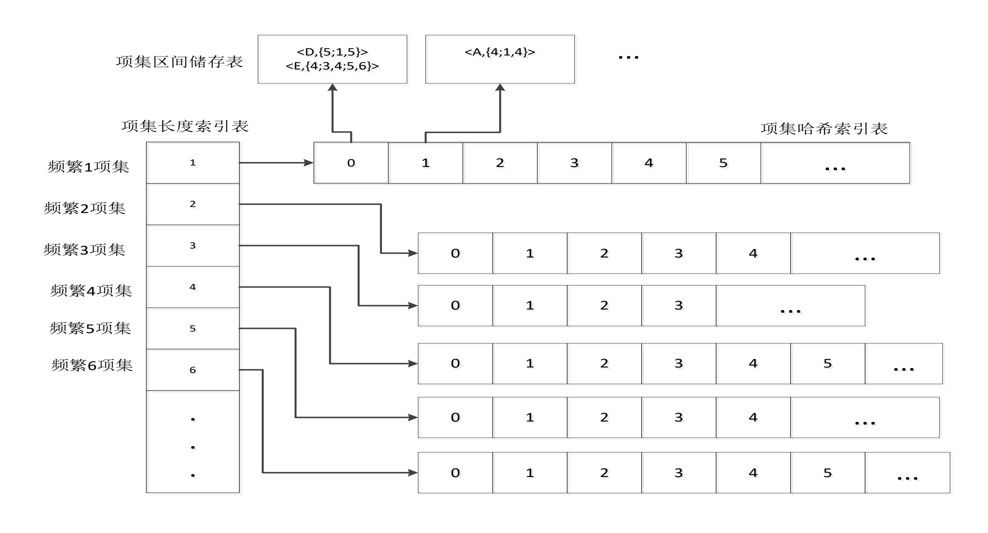


图 3 区间列表的存储结构图

3.2 候选项集的生成

IITM 算法采用区间来储存项集在数据集当中的分布信息，因此 IITM 算法只能采用 Apriori 类算法的方式来获取候选项集。Apriori 类算法获取候选项集的方式有大致两种，一种是使用两个频繁 n-1-项集来生成候选 n-项集，更具体地使用只有最后一项不相同的两个频繁 n-1-项集来生成候选 n 项集。例如有频繁 2-项集 ab,ac,bc,ad;生成候选项集时使用 ab 和 ac,ab 和 ad，ac 和 ad 来生成，生成的候选 3 项集为 abc,abd,acd;另一种是使用频繁 n-1-项集和频繁 1-项集合并来生成候选 n 项集，其中频繁 1 项集按照支持度递减的顺序排列，合并时使用频繁 n-1-项集和支持度小于其最后一项的频繁 1 项集来生成候选 n-项集，例如频繁 1-项集为 a,b,c,d 频繁 2-项集为 ab,ac,bc,那么生成的候选 3-项集为 abc,abd, acd, bdc;前一种方法生成的候选项集的数量要少于后一种，但是大多数情况下频繁 n-1-项集（n≥3）要比频繁 1-项集大得多，在频繁 n-1-项集当中搜索出只有最后一项不相同的两个项集也是不小的一笔开销。对于一般的 Apriori 类算法，减少一个候选项集就意味着减少一次对整个数据集的扫描，当数据集相当大的时候候选项集生成带来的消耗也就微不足道了，因此前一种算法会更优。但是对于 IITM 算法，减少一个候选项只是意味着少做一个区间集的求交，这时候当频繁 n-1-项集非常大的时候后一种方法的效率反而会更高。因此在 IITM 算法当中使用后一种方法。

3.3 区间求交

与 PIETM 算法不同 IITM 算法采用区间求交的方式来生成候选项集的区间。从 3.1 节中描述的构造过程可以得知，区间存储的其实是包含其对应项集的事务的所有的编号的集合，在该集合当中连续的编号被压缩存储在区间结构当中。需要求取的项集的支持度即是该区间集当中的事务编号的个数，所以一旦获取到区间集就能够获取到其对应的项集的支持度。在 IITM 算法通过频繁 n-1 项集和频繁 1 项集来生成候选 n-项集（n≥2）。设有频繁 n-1-项集 A,频繁 1-项集 B，它们合并生成的候选 n-项集为 C。C 对应的区间即同时包含 A 中所有的项和 B 中所有的项的事务编号的集合，即 A 对应的区间和 B 对应的区间的交集。

设{A}={supportA,[$a_{1\text{-start}}$,$a_{1\text{-end}}$],[$a_{2\text{-start}}$,$a_{2\text{-end}}$]… [$a_{n\text{-start}}$, $a_{n\text{-end}}$]}为 A 对应的区间集,{B}={supportB,[$b_{1\text{-start}}$,$b_{1\text{-end}}$][$b_{2\text{-start}}$,$b_{2\text{-end}}$]…[$b_{m\text{-start}}$,$b_{m\text{-end}}$]}为 B 对应的区间集通过分析发现区间求交的过程中当对区间集{A},{B}求交时，没有必要将{A}当中每个区间都去和{B}当中每个区间都去求交，因为{A},{B}当中的区间都是递增的，当满足一定条件时其实就可以判断{B}当中的剩余的区间中没有和该{A}区间有交集的区间,就可以避免很多次没有必要的求交操作。当使用{A}当中的一个区间 A_i=[$a_{i\text{-start}}$,$a_{i\text{-}}$

$_{end}$]去和{B}当中的一个区间 B_j=[$b_{j\text{-}start}$,$b_{j\text{-}end}$]求交时的方法如下:

若 $a_{i\text{-}start}$>$b_{j\text{-}end}$,A_i、B_j 没有交集，使用 A_i 去和 B_j 的下一个区间 B_{j+1} 进行求交,并记录{A}的下一个区间和{B}进行求交的起始位置 next=j+1。若 $a_{i\text{-}end}$<$b_{j\text{-}start}$，{B}中剩下的区间都比 $a_{i\text{-}end}$ 大，A_i 的求交过程完成，使用{A}当中的下一个区间 A_{i+1}=[$a_{(i+1)\text{-}start}$,$a_{(i+1)\text{-}end}$]来和{B}求交，并且可以跳过{B}当中的前面的区间，直接从 B_{next}=[$b_{next\text{-}start}$,$b_{next\text{-}end}$]这个区间开始求交，因为 $a_{i\text{-}start}$>$b_{j\text{-}end}$,$a_{(i+1)\text{-}start}$>$a_{i\text{-}end}$≥$a_{i\text{-}start}$,所以 $a_{(i+1)\text{-}start}$>$b_{j\text{-}end}$。因为{B}当中区间是递增的所以 B_j 区间以及其前面的区间都小于 $a_{(i+1)\text{-}start}$,因此不可能和区间 A_{i+1} 有交集，则可以直接从 B_{j+1}=[$b_{(j+1)\text{-}start}$,$b_{(j+1)\text{-}end}$]即 B_{next} 开始求交。

区间集生成之后查找 4.1 当中介绍的项集长度索引表判断该长度的项集的项集哈希索引表是否已经生成，如果没有生成则为该长度的项集生成项集哈希索引表，并且将生成的区间集插入到哈希索引表指定的项集区间储存表当中，如果存在则直接插入。

区间求交的核心思想伪代码如下：

```
输入：待求交的两个区间集 a,b
输出:两个区间求交得到的区间
merageIntervals（a,b）
  mergeList ={support=0;}//初始化合并结果区间列表
  next=1,i=1
  for a[i]=（ai-start,ai-end） in a //从第一个区间开始遍历 a 的每个区间
    for b[j=next]=(bj-start,bj-end) in b //从第 next 个区间开始遍历 b 的每个区间
      if(ai-start>bj-end)
        next = j+1//更新 next 值
        continue;
      else if( ai-end<bj-start)
        break;//使用 a 当中的下一个区间和 b 直接从第 b 中的第 next 个区间开始求交
      intersection a[i] and b[j] save into mergList//将区间 a[i]和 b[j]求交并将结果保存的 mergList 当中
      update support in mergList //更新支持度计数
  output mergeList//输出区间集求交结果
```

3.4　IITM 算法的总结

IITM 算法流程如图 4 所示，首先使用 2.2 节中的方法生成 FP 树，然后使用 3.1 节中的方法将 FP 树当中的各个项映射到区间当中。接着使用生成 FP 树时得到的频繁 1 项集来进行模式增长。首先对频繁 1-项集按支持度递减排序,然后遍历频繁 1-项集,使用每个项和排在它后面的项进行结合生成候选 2-项集,接着使用这两个项对应的区间进行求交生成候选 2-项集的区间并且得到候选 2-项集的支持度来判断该候选 2-项集是否需要剪枝。频繁 2-项集生成后，使用频繁 2-项集和频繁 1-项集合并来生成频繁 3-项集，与生成频繁 2-项集时类似，遍历频繁 2-项集，对每个频繁 2-项集每个项，将它分别与频繁 1-项集中排在该 2-项集的最后一个项后面的项相结合来生成候选 3-项集。使用同样的方法来使用频繁 k-项集来生成频繁 k+1-项集直到不再能产生频繁项为止。使用表 2 当中的频繁 1 项集生成的候选 2-项集如表 3 所示。

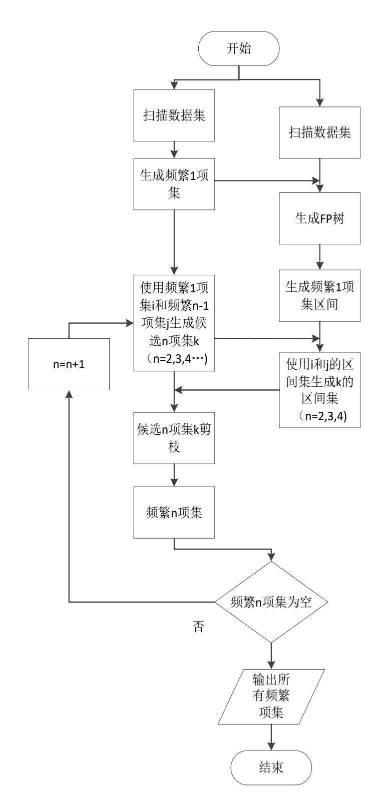


图 4　IITM 算法的流程图

表 3　根据表 2 当中的频繁 1 项集生成的候选 2-项集及区间

项名	支持度	区间
DE	3	[3,5]
DA	4	[1,4]
DB	2	[1,2]
EA	2	[3,4]
EB	1	[6,6]
AB	2	[1,2]

4　实验以及结果

4.1　实验数据集

本文采用四个真实数据集来进行实验，实验数据集的基本信息如表 4 所示

表 4　实验数据集信息

数据集名称	事务数量	事务长度	密集程度
Mushroom	8124	22	较密集
Connect	67557	42	稀疏
Chess	3196	37	密集
Retail	88162	非定长	稀疏

Mushroom 数据集能够从 UCI 机器学习仓库当中获得（http://archive.ics.uci.edu/ml/），Mushroom 数据集当中不同属性的值在数据集当中显示的标示可能是一样的，因此本文在开始实验之前本文为数据集当中的每个项加上它对应的属性标号。例如对于一条数据：p,x,s,n,t,p,f,c,n,k,e,e,s,s,w, w,p ,w, o,p, k,s,u 本文为它加上属性编号之后为 1p,2x,3s,4n, 5t, 6p,7f, 8c,9n,10k,11e,12e,13s,14s,15w, 16w,17p,18w,19o, 20p,21k, 22s,23u.Connect,Chess,Retail 数据集可以从（http://fimi.ua.ac. be/data/）中获取,直接使用数据来进行实验。

4.2　实验环境

实验当中的算法都使用 Java 在 eclipse 当中实验。实验的硬件环境为 2.4 GHz CPU,4 GB 运行内存。

4.3　实验结果

本文实现了 Apriori,FP-Growth,PIETM,以及 IITM 算法。在 Mushroom 数据集上在支持度为 10%，15%，20%，25%，30%，35%，40%时获取 PIETM 和 IITM 算法的运行时间，其中因为 Apriori 算法在支持度较低的情况下执行效率较低，因此仅仅在 25%到 40%的支持度进行了实验。由于算法的执行时间随支持度变化较大,因此将执行时间采用对数坐标。实验结果如图 5 所示，从实验结果当中可以看出 FP-Growth、PIETM 和 IITM 算法要明显优于 Aproiri 算法。当支持度小于 25%的时候从图中可以看出 IITM 算法的执行时间要明显优于 PIETM 和 FP-Growth。支持度较高时两种算法差别不明显，因此补充了在较高支持度下 Mushroom 数据集上两种算法的比较。实验的支持度范围在 30%~40%，每隔 2%设置一个实验点。在支持度较高时算法的运行时间较短容易受偶然因素的影响，因此本文每个点的实验共进行 5 次，然后 5 次实验得到的结果取平均值来作为该点的实验结果。各点的实验结果如图 6 所示，从图中可以看出，当支持度较高时，IITM 算法也会优于 PIETM 算法以及 FP-Growth 算法。

由于 Connect 数据集的数据量较大，本文只做了支持度较高环境下的实验。因为 Aproiri 算法的效率过低，只用 PIETM 和 IITM 以及 FP-Growth 算法来来进行比较，本文获取了支持度为 80%、85%、90%、95%时这两个算法运行时间。具体的实验结果如图 7 所示，从图中可以看出 IITM 的运行时间要优于 PIETM 和 FP-growth。

Chess 数据集数据分布密集，本文只在较高支持度上对 PIETM 和 IITM 以及 FP-Growth 算法进行了实验对比，Retail 数据集是高度稀疏的数据集，本文在低支持度下对三种算法进行了实验对比，实验结果表明在两个数据集上，不同支持度下 IITM 算法都要优于另外两个算法。具体实验结果分别如图 8、9 所示。

5　结束语

本文提出了频繁模式挖掘算法 IITM,该算法是在 PIETM 算法的基础之上进行改进的。该算法参考 PIETM 算法的事务映射思想将 FP 树当中的各个项映射到区间当中，并提出一种新的区间集存储结构来提高区间集的查找效率，同时该算法提出了一种新的候选项集的区间集以及其支持度的获取方式：通过求交获取候选项集的区间集，从区间集直接获取候选项支持度。该方式要比 PIETM 算法当中的通过求并获取候选项集的区间集而后通过容斥原理获取其支持度的效率要高。IITM 算法还提出了一种快速的区间集求交方法来减少区间集求交过程当中的冗余操作，进一步提高了效率。

本文在多个数据集上对 Apriori、FP-Growth、PIETM、IITM 四个算法进行了比较，实验结果表明在不同的支持度下 IITM 算法的效率都要高于 Apriori、FP-Growth 以及 PIETM 算法。

下一步的工作包括研究该算法的并行化实现，扩展到 Hadoop、Spark 等大数据处理平台中。

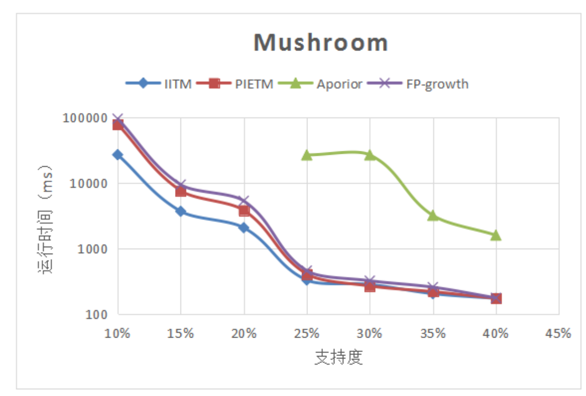


图 5　Mushroom 数据集上的实验结果

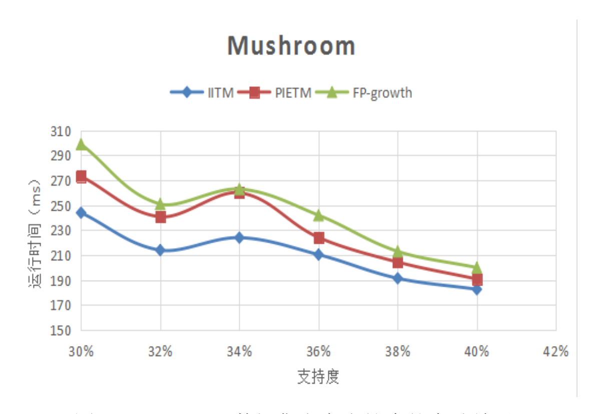


图 6　Mushroom 数据集上高支持度的实验结果

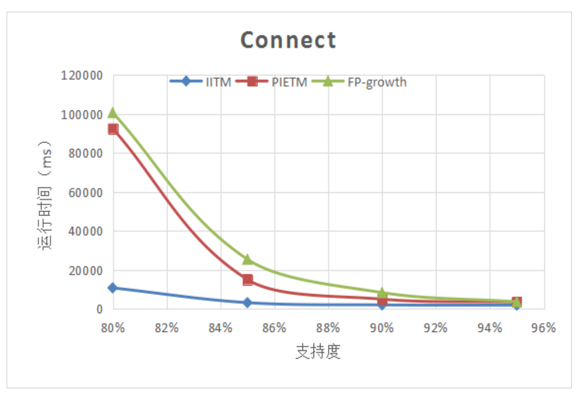


图 7　Connect 数据集上的实验结果

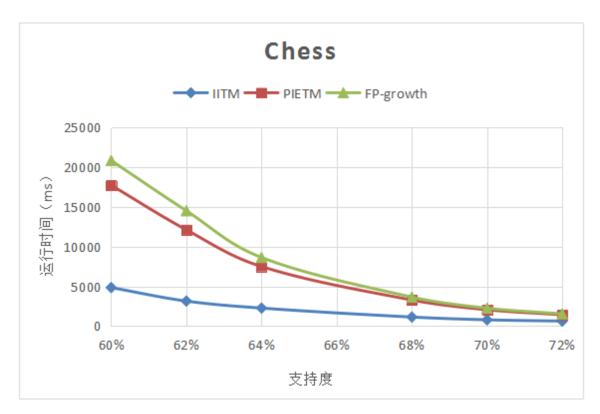


图 8　Chess 数据集上的实验结果

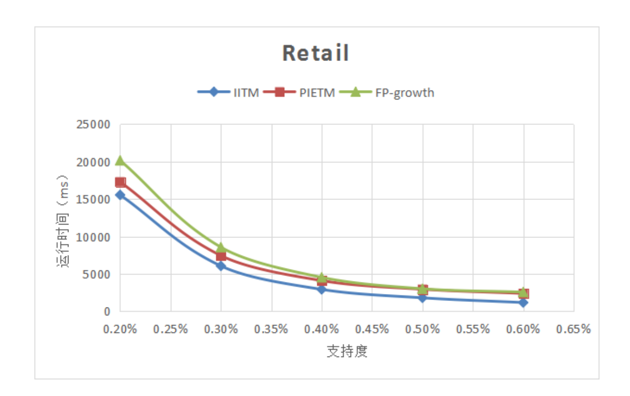


图 9　Retail 数据集上的实验结果

参考文献：

[1] Agrawal R, Imieliński T, Swami A. Mining association rules between sets of items in large databases [C]// Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 1993: 207-216.

[2] Lee G, Yun U, Ryu K H. Sliding window based weighted maximal frequent pattern mining over data streams [J]. Expert Systems with Applications, 2014, 41 (2): 694-708.

[3] Zou J, Xiao J, Hou R, et al. Frequent instruction sequential pattern mining in hardware sample data [C]// Proc of IEEE International Conference on Data Mining. 2010: 1205-1210.

[4] Chen Y C, Peng W C, Lee S Y. CEMiner: an efficient algorithm for mining closed patterns from time interval-based data [C]// Proc of International Conference on Data Mining. Washington DC: IEEE Computer Society, 2011: 121-130.

[5] Han J, Jian P, Yin Y, et al. Mining frequent patterns without candidate generation: a frequent-pattern tree approach [J]. Data Mining & Knowledge Discovery, 2004, 8 (1): 53-87.

[6] Lin K C, Liao I E, Chang T P, et al. A frequent itemset mining algorithm based on the principle of inclusion-exclusion and transaction mapping [J]. Information Sciences, 2014, 276 (C): 278-289.

[7] Park J S, Chen M S, Yu P S. Using a hash-based method with transaction trimming for mining association rules [J]. IEEE Trans on Knowledge & Data Engineering, 1997, 9 (5): 813-825.

[8] 刘步中. 基于频繁项集挖掘算法的改进与研究 [J]. 计算机应用研究, 2012, 29 (2): 475-477.

[9] Li N, Zeng L, He Q, et al. Parallel implementation of apriori algorithm based on MapReduce [C]// Proc of International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing. 2012: 236-241.

[10] Rathee S, Kaul M, Kashyap A. R-Apriori: an efficient apriori based algorithm on Spark [C]// Proc of Ph. d. Workshop in Information and Knowledge Management. 2015: 27-34.

[11] Ye F Y, Wang J D, Shao B L. New algorithm for mining frequent itemsets in sparse database [C]// Proc of International Conference on Machine Learning and Cybernetics. 2005: 1554-1558.

[12] Lin C W, Gan W, Fournier-Viger P, et al. Weighted frequent itemset mining over uncertain databases [J]. Applied Intelligence, 2016, 44 (1): 1-19.

[13] Lin K C, Liao I E, Chen Z S. An improved frequent pattern growth method for mining association rules [J]. Expert Systems with Applications, 2011, 38 (5): 5154-5161.

[14] Grahne G, Zhu J. Fast algorithms for frequent itemset mining using FP-trees [J]. IEEE Trans on Knowledge & Data Engineering, 2005, 17 (10): 1347-1362.

[15] 杨鹏坤, 彭慧, 周晓锋, 等. 改进的基于频繁模式树的最大频繁项集挖掘算法 FP-MFIA [J]. 计算机应用, 2015, 35 (3): 775-778.

[16] Song Y G, Cui H M, Feng X B. Parallel incremental frequent itemset mining for large data [J]. Journal of Computer Science & Technology, 2017, 32 (2): 368-385

[17] Lin C W, Gan W, Fournier-Viger P, et al. Efficient algorithms for mining recent weighted frequent itemsets in temporal transactional databases [C]// Proc of ACM Symposium on Applied Computing. 2016: 861-866.

[18] Lin C W, Gan W, Fournier-Viger P, et al. Efficient Mining of Weighted Frequent Itemsets in Uncertain Databases [M]// Machine Learning and Data Mining in Pattern Recognition. [S. l.] : Springer International Publishing, 2016.

[19] 曹博, 倪建成, 李淋淋, 等. 基于 Spark 的并行频繁模式挖掘算法 [J]. 计算机工程与应用, 2016, 52 (20): 86-91.